MVC, MVP, MVVM pattern

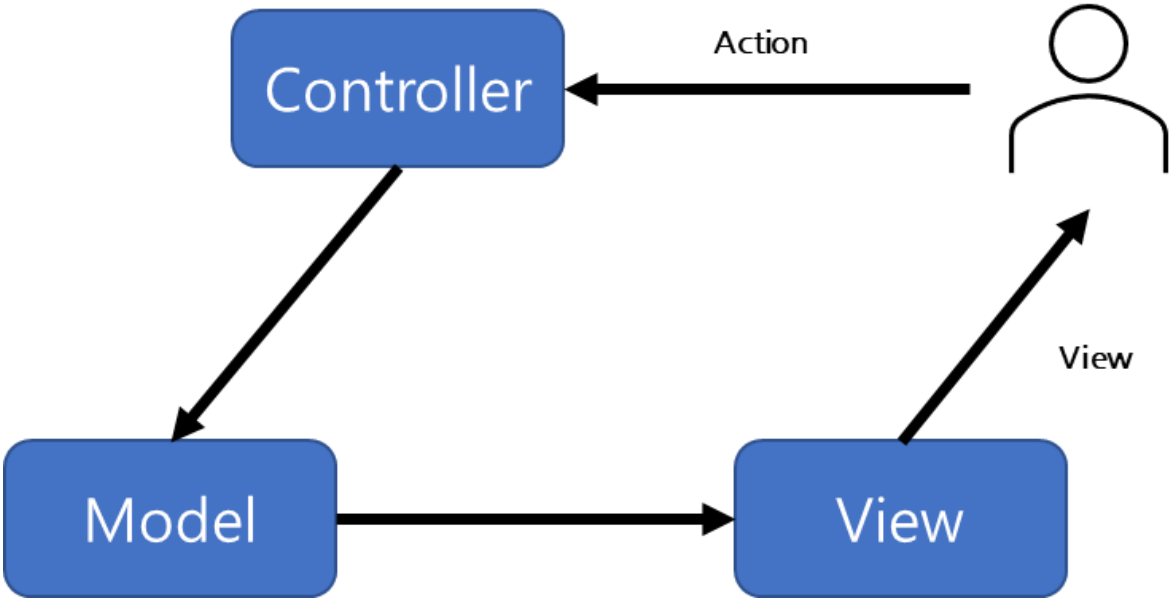
소프트웨어 개발 분야에서 자주 쓰이는 UI 디자인 패턴들.

시간 순으로는 MVC 패턴이 1970년대 후반에 만들어졌고
이후 HMVC, MVA, MVP, MVVM이라는 패턴들이 파생되었음.

> UI 개발자는 백엔드 데이터와 시각 디자인을 별개의 시스템으로 나눕니다. 이렇게 하면 UI를 빌드하는 프로세스가 변경사항으로 인한 영향을 덜 받게 됩니다. 디자이너는 UI 이면의 코드를 변경하는 일 없이 시각적 프레젠테이션을 변경할 수 있고 프로그래머는 프런트엔드가 완료되기를 기다릴 필요 없이 데이터와 시스템에 집중할 수 있기 때문입니다.

from EpicGames.

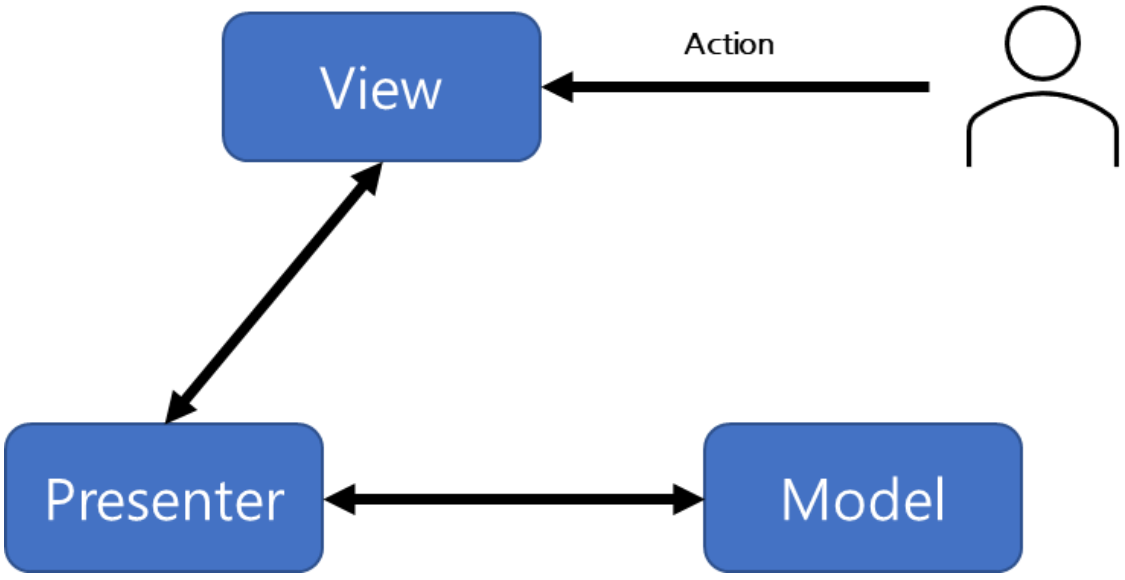
MVC(Model - View - Controller)



- **Model:** 데이터와 데이터에 관련된 로직. 쉽게 생각하면 데이터 관리자.
  (Unreal에서는 CSV 파일, 능력치나 스킬 등의 데이터를 관리하는 클래스)
- **View:** Model을 시각적으로 보여줌. 그냥 UI.
- **Controller:** 입력을 받고 이를 기반으로 Model을 조작(업데이트)하고, Model을 나타내어줄 View를 선택.

이때 Controller는 View를 선택(1:N)만 하고 업데이트를 시켜주지 않기 때문에 View는 Model을 이용하여 업데이트 하게 됨. 이와 같이, **View**는 **Model**을 이용하기 때문에 **서로 간의 의존성을 완벽히 피할 수 없다는 단점**이 존재.

MVP(Model - View - Presenter)



MVC 패턴에서의 Mode과 View는 동일한 역할.

- **Presenter:** 입력을 받고 이를 기반으로 Model와 Veiw를 업데이트. 중개자 역할 수행.

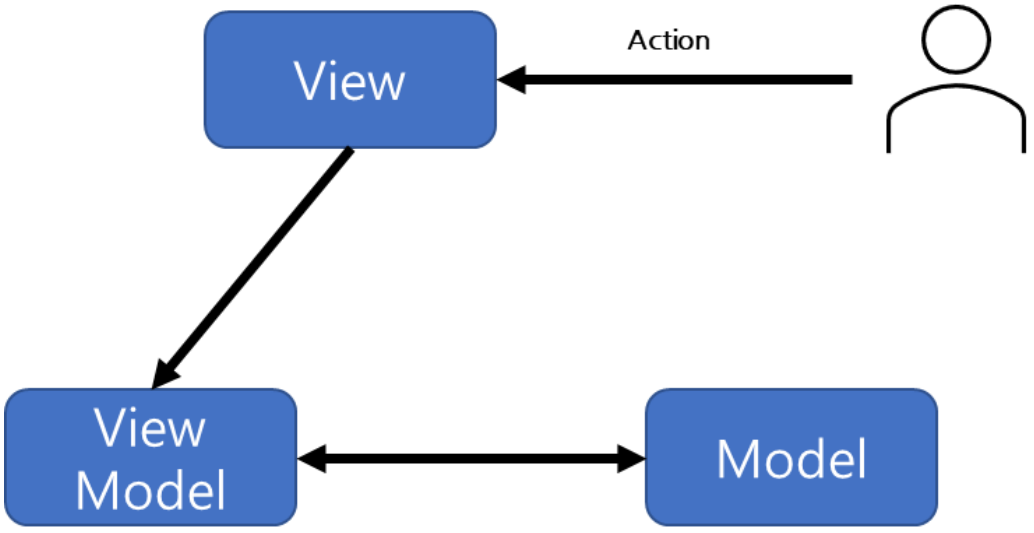
MVC 패턴과는 다르게 Presenter를 통해 Model과 View를 완벽히 분리해 주기 때문에 View는 Model을 따로 알고 있지 않아도 된다는 장점이 있음. 단점으로는 View와 1:1 관계이기 때문에 View와의 의존성이 매우 강하다.

```
// Model
public class Item
{
    public string Name { get; set; }
```

```
    public int Price { get; set; }
}

// View
public interface IShopView
{
    void UpdateUI(Item item);
}

public class ShopView : MonoBehaviour, IShopView
{
    public Text itemText;
    public Text priceText;
    public Button buyButton;
    private ShopPresenter shopPresenter;

    private void Awake()
    {
        shopPresenter = new ShopPresenter(this);
    }

    public void UpdateUI(Item item)
    {
        itemText.text = item.Name;
        priceText.text = item.Price.ToString();
    }

    public void OnBuyButtonClick()
    {
        shopPresenter.BuyItem();
    }
}

// Presenter
public class ShopPresenter
{
    private IShopView shopView;
    private Item currentItem;
    private List<Item> playerInventory = new List<Item>();

    public ShopPresenter(IShopView view)
    {
        shopView = view;
    }

    // 상점 초기화 메서드
    public void InitializeShop()
    {
        // 데이터베이스나 다른 소스에서 아이템 데이터를 로드합니다.
        currentItem = new Item { Name = "검", Price = 100 };

        // 현재 아이템으로 UI 업데이트
        shopView.UpdateUI(currentItem);
    }

    // 아이템 구매 메서드
```

```
    public void BuyItem()
    {
        // 플레이어의 통화량을 확인합니다.
        int playerCurrency = GetPlayerCurrency(); // 플레이어의 통화량을 가져오는 메서드
        if (playerCurrency >= currentItem.Price)
        {
            // 아이템 가격을 플레이어의 통화량에서 차감합니다.
            playerCurrency -= currentItem.Price;
            UpdatePlayerCurrency(playerCurrency); // 플레이어의 통화량을 업데이트하는 메서드

            // 아이템을 플레이어의 인벤토리에 추가합니다.
            AddItemToInventory(currentItem); // 아이템을 플레이어의 인벤토리에 추가하는 메서드

            // 구매 성공 메시지 표시 또는 다른 작업 수행
            Debug.Log("아이템을 성공적으로 구매했습니다!");
        }
        else
        {
            // 에러 메시지 표시 또는 다른 작업 수행
            Debug.Log("통화량이 부족하여 아이템을 구매할 수 없습니다!");
        }
    }

    private int GetPlayerCurrency()
    {
        // 플레이어의 통화량을 가져오는 로직 구현
        return 0;
    }

    private void UpdatePlayerCurrency(int currency)
    {
        // 플레이어의 통화량을 업데이트하는 로직 구현
    }

    private void AddItemToInventory(Item item)
    {
        playerInventory.Add(item);
    }
}
```

MVVM(Model - View - Viewmodel)



앞 패턴들에서 말했던 Model과 View 역할과 동일. MVP와 같이 View에서 입력이 처리된다.

- **ViewModel**: View를 표현하기 위해 만들어진 View를 위한 Model. View가 사용할 함수와 필드를 구현하고, VIew에게 상태 변화를 알림.

Command 패턴과 Data Binding으로 MVP 패턴과 달리 View와의 의존성을 완벽히 분리 할 수 있다는 장점이 있다.

```
// Model
public class Item : INotifyPropertyChanged
{
    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                name = value;
                OnPropertyChanged(nameof(Name));
            }
        }
    }

    private int price;
    public int Price
    {
        get { return price; }
        set
        {
            if (price != value)
            {
```

```
                price = value;
                OnPropertyChanged(nameof(Price));
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}

// View
public class ShopView : MonoBehaviour
{
    public Text itemText;
    public Text priceText;
    public Button buyButton;

    private Item currentItem;

    public void Initialize(Item item)
    {
        currentItem = item;
        UpdateUI();
    }

    private void UpdateUI()
    {
        itemText.text = currentItem.Name;
        priceText.text = currentItem.Price.ToString();
    }

    public void BuyItem()
    {
        // 플레이어의 통화량을 확인합니다.
        int playerCurrency = GetPlayerCurrency(); // 플레이어의 통화량을 가져오는 메서드
        if (playerCurrency >= currentItem.Price)
        {
            // 아이템 가격을 플레이어의 통화량에서 차감합니다.
            playerCurrency -= currentItem.Price;
            UpdatePlayerCurrency(playerCurrency); // 플레이어의 통화량을 업데이트하는 메서드

            // 아이템을 플레이어의 인벤토리에 추가합니다.
            AddItemToInventory(currentItem); // 아이템을 플레이어의 인벤토리에 추가하는 메서드

            // 구매 성공 메시지 표시 또는 다른 작업 수행
            Debug.Log("아이템을 성공적으로 구매했습니다!");
        }
        else
        {
            // 에러 메시지 표시 또는 다른 작업 수행
            Debug.Log("통화량이 부족하여 아이템을 구매할 수 없습니다!");
        }
```

```
    }
}

// ViewModel
public class ShopViewModel : MonoBehaviour
{
    public ShopView shopView;
    public Item currentItem;

    // 상점 초기화 메서드
    public void InitializeShop()
    {
        // 데이터베이스나 다른 소스에서 아이템 데이터를 로드합니다.
        currentItem = new Item { Name = "검", Price = 100 };

        // View에 ViewModel 설정
        shopView.Initialize(currentItem);
    }
}
```

💡 **ViewModel과 MVP 패턴의 Presenter의 주요 차이점**: Presenter는 View에 대한 참조를 갖고 있는 반면 ViewModel은 참조하지 않음. 대신, View는 ViewModel에 직접 선언되어 업데이트를 주고 받음.

언리얼의 UMG 뷰모델(베타)

UMG 뷰모델

뷰모델을 사용하여 이벤트를 처리하고 UI에서 변수의 매니페스트를 관리하는 방법을 살펴봅니다.

https://docs.unrealengine.com/5.2/ko/umg-viewmodel/

5.3 공식 문서는 아직 UMG 뷰모델 목차가 쓰여지지 않았는데, 버린 건지 아직 안 쓴 건지 모르겠다.